A Voxel-Based, Forward Projection Algorithm for Rendering Surface and Volumetric Data

John R. Wright
Julia C. L. Hsieh
Hughes Training, Inc.

Abstract

In this paper we present a voxel-based, forward projection algorithm with a pipeline architecture for real-time applications. The multi-sensor capabilities (electro-optical, or visual, and infrared) currently implemented in software have also been applied to non-real-time imaging applications on workstations and minicomputers. Most suited for terrain based applications, the system features haze, imbedded targets, moving objects, smooth shading, and specular reflections.

Introduction

For the past several years our group at Hughes Aircraft Co., and later Hughes Training, Inc., has been developing voxel-based algorithms for the rendering of surface and volumetric data. This work was originally intended to be used in the construction of real-time image generation systems for use in flight simulators and other demanding applications. It has since grown to include non-real-time imaging applications on workstations and minicomputers. Some of our earlier work was similar to methods presented by Devich and Weinhaus[1] but has been enhanced to offer true perspective from any viewing angle. The rendering algorithm provides several features applicable to flight simulators, such as haze, repositionable target models, smooth shading, and specular reflections, and is designed for implementation on a pipelined processor for real-time applications. In addition, the terrain and object databases are implemented using voxels, or volume elements, to provide greater resolution and scene content than is normally available in polygon-based systems. The eyepoint is specifiable with six degrees of freedom as are the positions of the objects. The rendered image is a perspective projection of the voxels to the screen along rays from the database to the eyepoint. This puts it in the class of forward projection algorithms with similarities to the cell by cell processing discussed by Upson and Keeler [2]. Multiple objects are processed from back to front and range resolved with a modified Z-buffer.

Definitions

The Hughes rendering algorithm uses two types of voxels, called case II and case III. A case II voxel is used to represent terrain and surface data and is a rectangular solid represented by four corner posts with a surface stretched between them. Each corner has an accompanying height and color which are bilinearly interpolated across the surface. The corner posts are shared between the four neighboring voxels so the surface is contiguous. Normally, only the top of a case II voxel is visible. If the side is visible, it will be a constant color. For vertical surfaces with some variety, case III voxels are used. A case III voxel contains a height and a color, which represent a base, and a pointer to a vertical stack of segments, called a pattern, which make up the vertical information resting on top of the base. Each segment in the stack contains a height, or length, and a left and right color. The left and right colors provide for interpolation in situations where color varies across the volume of the voxel. It also contains surface normal information, for shading and specularity, and some flags. The segments may be opaque, transparent, or translucent. In fact, all voxels can have an opacity anywhere between transparent and opaque. All references to color later in this paper include an accompanying opacity and specularity. Repositionable objects, such as tanks, trucks, ships, aircraft, explosions, etc., are composed of case III voxels. A terrain database may contain embedded case III objects, such as buildings and trees, surrounded by case II voxels. Most of the case III objects we have used were voxellized (scan-converted to voxels) from polygon models using techniques developed by Kaufman[3,4,5], and Cohen and Kaufman[6].

A database is composed of several layers, each containing a particular type of information such as elevation, color, flags, pattern numbers, etc. The layers are broken up into rectangular regions which represent an area of the database at a given resolution. All the layers within a region are correlated with each other through various orthorectification and image mosaicking steps during database creation. Many of the steps involved are presented by Whiteside[7]. Multiple resolution levels are generally available for each region providing for lower resolution data over a wide area with high resolution inserts for approaches and other low altitude operations. Each database also has a corresponding pattern segment table containing the patterns for all case III voxels in all regions in the database. Each database rests on a baseplane which is the (x,y) plane at an elevation (z) value of zero.

A database can be visual, with an RGB layer, or infrared (IR) with an intensity layer. During the creation process, an IR database begins as multi-spectral images. These multi-spectral images are segmented into areas of similar characteristics and the material types of the areas are determined. The thermal parameters, such as albedo, emissivity, and conductivity, are then used, along with the sun position and surface normal, to determine the thermal radiance for each voxel in the database. The radiance layer is then rendered the same as an RGB database. The process for generating these IR databases is discussed by Quarato, et al[8]. This method provides for correlated sensor databases. Similar databases can be used for synthetic aperture, real beam, or millimeter wave RADAR or LASER RADAR (LADAR) imaging. The algorithms for rendering would be somewhat different, however. Brown, et al[9], discuss the use of IR simulation algorithms for voxel target models.

The eyepoint, as well as the positions of moving objects, are specified with six degrees of freedom: x, y, z, roll, pitch, and yaw. The x, y, and z values are specified relative to the origin of the database (e.g. the southwest corner). Yaw is rotation about the z (vertical) axis measured relative to the x axis of the database. Pitch is rotation about the y axis above or below the horizon. Roll is rotation about the x axis which is the boresight. The boresight is defined as a line passing through the eyepoint and the center of the screen. Figure 1 illustrates these relationships.

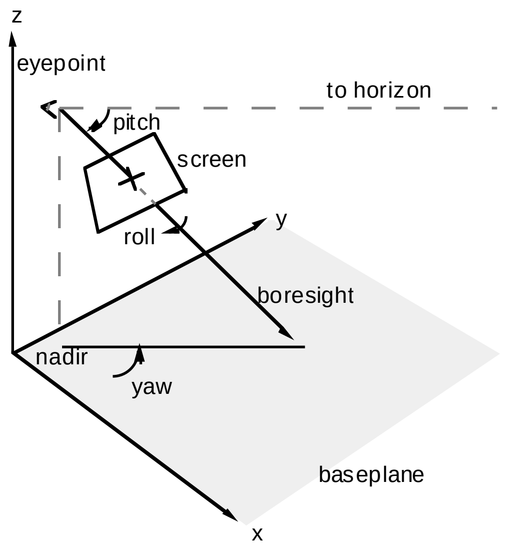

Figure 1 - Six Degrees of Freedom

Overview

The rendering algorithm is divided into two separate pipelineable sections, the voxel pipe and the pixel pipe. Each pipe consists of several modules which perform the appropriate calculations. The voxel pipe accesses the database voxels along a straight line which is the current column of pixels projected onto the baseplane of the database. This column of voxels is mapped onto the pixel column which is then processed by the pixel pipe into a column of pixels across the frame buffer. The projection of voxels to the screen, rather than the projection of pixels to the database, is the hallmark of a forward projection algorithm. Figure 2 illustrates the perspective projection of voxels to the screen.

Figure 2 - Perspective Projection and Resolution Selection
This figure is missing!!!

A column of voxels is generated by starting at the nadir, which is the point of the database immediately below the eye, and stepping out along a straight line. The steps are determined by projecting a pixel column to the baseplane and computing the angle between the line thus generated and the axes of the database. The delta x and delta y are found using the sine and cosine of the computed angle. This fixed pattern of accessing the database means that the rendering time is relatively independent of scene complexity. Each step lands on a voxel which may be either of the two types listed above. If the voxel is case II, the height and color are bilinearly interpolated from the four corners to the step point within the voxel. If the voxel is a case III voxel, the corresponding pattern is located and each segment in the pattern becomes another voxel in the column. At this point, each voxel consists of a height, color, and ground range from the eye. From the height of the voxel, the height of the eye, and the ground range to the step point, the depression angle to the top of the voxel is calculated. This is the angle above or below the horizon with the nadir being 90 degrees, the zenith being -90 degrees, and the horizon being zero.

Knowing the depression angle to the voxel, as well as the depression angle to the top and bottom of the screen for the current column, it is possible to calculate the fraction of the column being covered by the voxel. If a case III voxel is being processed, the depression angle of both the top and bottom of each pattern segment are used to calculate the coverage. The column is divided into pixels with each pixel occupying a fixed angular portion of the column. Therefore, a voxel is converted to pixels by computing which pixels, or portions of pixels, are covered by the voxel through simple angular comparisons. The voxels are accumulated into pixels in a front to back order.

Once the entire column of pixels has been generated it goes into the pixel pipe. The pixel pipe takes each angular pixel and calculates a screen, or frame buffer, location for it. This is done by using the depression angle of the pixel and the angle between the current column and the column at the center of the screen and doing a polar to flat screen coordinate transformation. At the same time, it performs the roll calculation, rotating the pixels to their final (x,y) position. The pixel pipe then performs a Z-buffer range resolution function by discarding pixels behind those already on the screen and then integrates the visible pixels into neighboring positions in the frame buffer.

It is important to note that the voxel columns are projected as planes passing through the eyepoint and the nadir such that they are always perpendicular to the baseplane of the database being rendered. This means that all segments of a vertical pattern will be projected into the same column. However, a column of pixels will not necessarily be parallel to any edge of the screen or even to any other column. In fact, the columns form a radial spoke pattern with the nadir or zenith at the center, as shown in Figure 3. The radial scan technique is similar to that presented by Patz, et al[10].

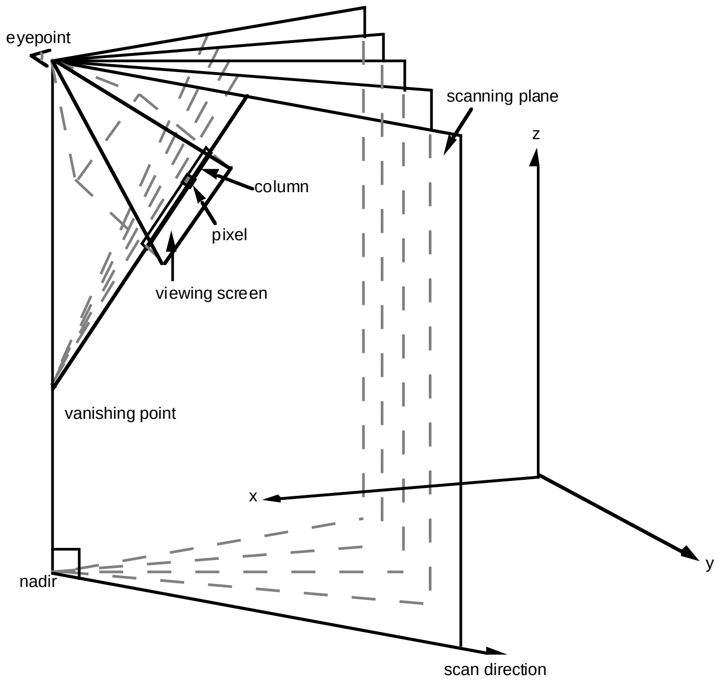

Figure 3 - Radial Scan

Details of the Algorithm

Figure 4 contains a block diagram of the modular structure of the rendering algorithm implementation.

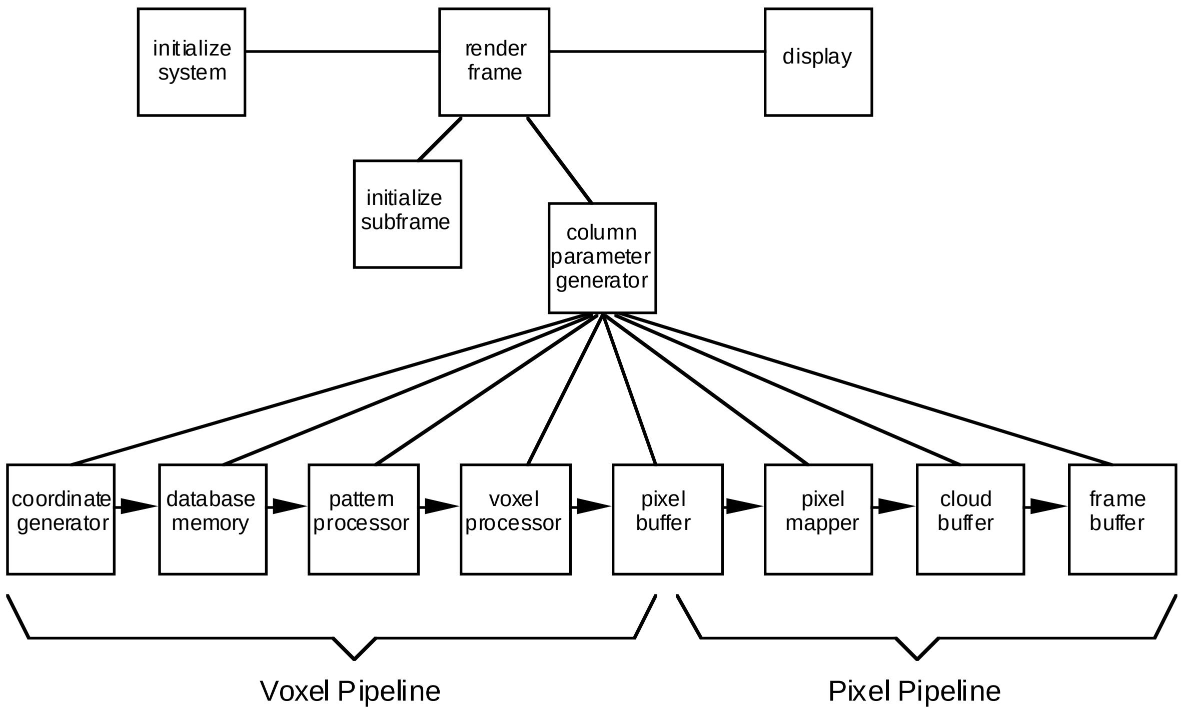

Figure 4 - Modular Composition of Rendering Algorithm

The modules included in the diagram will be discussed in more detail individually.

The Initialize System module initializes the system, loads the database, and initiates the frame rendering. The database load function, deserves a bit of explanation. It loads the various layers of the database, as well as the vertical pattern information. The basic function is to load a database region and to update the valid region tables for the Coordinate Generator module. Note that the regions must be rectangular but may be of any size. Multiple regions, which may partially or fully overlap, may be loaded for a given resolution level. The valid region tables contain the information about what regions, at what resolution levels, are valid for which areas of the database.

The Render Frame module performs some frame initialization calculations, reads in the eye and object positions for the frame, and then renders the subframes by initiating the appropriate modules. In particular, the frame initialization includes sorting the moving objects based on range. The subframes are rendered terrain first (if any), and then any moving objects from far to near. This ordering is important because the pixels must enter the frame buffer in a far to near order for proper accumulation and attenuation if translucencies are involved. The Initialize Subframe module then performs the subframe initialization tasks such as determining object visibility, scan limits, trim angles, and pixel subtend angle, and transforming the eye position in world coordinates to the eye position in subframe coordinates. The scan limits calculation includes checking to see if the subframe should be clipped on the frame boundary or the object's boundary or both. In order to insure that all pixels on the screen are generated, yet minimize the number of columns and pixels per column, it is desirable to trim the columns at the edges of the screen, or at the limits of the object, prior to generating the pixels. To do this, it is necessary to use the screen parameters (width, height, field of view, pitch, roll, etc.) to generate the appropriate column parameters.

The rendering of a single frame is broken down into the rendering of multiple subframes. Each subframe consists of rendering a single database, meaning the terrain, or a single object. The rendering of a subframe is done under the control of the Column Parameter Generator (CPG) module. The CPG generates various parameters for the processing of each of the columns needed to complete a subframe within the scan limits previously determined. These are the sine and cosine of the ground column angle (the angle between the projection of the column to the baseplane and the projection of the boresight column), the angle to the bottom and top of the screen (or trim angle) for that column, the delta x and delta y and the diagonal step size. Given the ground column angle, the CPG computes the pitch angle to the top and bottom of the screen for the particular column. Because of the radial scan and roll, the column may actually intersect the sides of the screen instead of top and bottom. The CPG determines which sides of the screen the column intersects and then applies the formula:

$$P = \tan^{-1} \frac{\sin P * \cos(-col) * \cos R + \sin(-col) * \sin R - \cos P * \cos(-col) * \tan(FOV/2)}{\cos P * \cos R + \sin P * \tan(FOV/2)}$$

where P = pitch, R = roll, FOV = field of view measured horizontally across the width of the screen, and col = ground column angle. This formula gives the pitch of the intersection of the column and the top of the screen. Similar calculations, which also involve the aspect ratio of the screen, compute the pitch for other edges. If the nadir is on the screen, then the bottom angle is set to +90 degrees. Similarly, if the upper nadir, or zenith, is on the screen, then the top angle is set to -90 degrees.

The Voxel Pipe

The Coordinate Generator module uses an initial x and y starting position and a delta x and y to generate a series of voxel addresses which step across the database. For each x and y step, the Coordinate Generator checks a valid region table to find the best resolution data available at that (x,y) position. The best resolution is the finest resolution data available which is at least as coarse as the desired resolution level, determined by slant range. Once it has determined the resolution level of the best data available, it locates the address of that data within the database memory.

The Database Memory module accepts a column of voxel addresses and accesses its memory to retrieve the color and elevation information for the four neighbor corners. If the voxel is of type case II, the four corners are then bilinearly interpolated in color and elevation and a final voxel is produced. If the voxel is III, the appropriate data for those voxel types is also retrieved.

The Pattern Processor module handles the voxels which are case III. The case III voxels cause the Pattern Processor module to access the pattern memory to retrieve the vertical color and elevation information above the base voxel. Each piece of vertical information, called a segment, becomes a separate voxel within the voxel arrays. To ensure the proper accumulation of voxels into pixels, the Pattern Processor module will output the stack of segments from the eye elevation up and then down. This ensures that all voxels will contribute to pixels from the nearest to the farthest. Within the voxel/pixel column, the voxels are processed from front to back. Certain flags must be set to ensure that the voxels going up and those going down are processed appropriately. In addition, the Pattern Processor performs smooth shading calculations for the case III voxels. Each case III pattern segment has a horizontal and a vertical orientation value. Using the sun position, the Pattern Processor computes the incident sun angle for each segment and combines the sunlight color, the ambient light color, and the voxel color using the following formula:

$$I_{final} = I_{vox} * (I_{amb} + I_{sun} * \cos(incident\_angle))$$

The formula is computed for each channel, once for IR or three times for RGB. This is a standard formula as discussed by Foley and Van Dam[11].

The Voxel Processor accepts a stream of voxels which have been prepared by the Pattern Processor and calculates the depression angle and slant range to each. The slant range and depression angle are calculated from the ground range from the eye to the voxel (as determined by the number of steps taken to get there), the height of the voxel above the base plane, and the height of the eye above the base plane. Voxels are processed as rectangular solids so they have the appropriate cross-section at any viewing angle.

The other major function of the Voxel Processor is the planning scan. When the pipeline processor begins to process a column, it begins by doing a planning scan at a coarse resolution to determine the correct hierarchies to access for each region. The Voxel Processor computes the desired hierarchies during the planning scan. The correct hierarchy is determined by requiring the system to choose voxels which are wider than the distance between neighboring columns at that slant range. This causes the algorithm to select coarser and coarser resolution data as it renders data further from the eyepoint. To find this distance the following formula is used:

distance = sin(column_angle) * slant_range

Once this distance is known the appropriate size voxels can be selected. Figure 2 illustrates the selection of appropriate sized voxels based on slant range. To avoid a large processing overhead for the planning scan, the voxel data several levels of resolution coarser than expected shall be used for the plan, if available. This method allows us to determine the desired hierarchy for the voxels but does not allow us to locate hidden regions or determine the number of steps to the first voxel visible on the screen.

The Pixel Buffer is one of the most sophisticated functions in the renderer. It accepts voxels from the Voxel Processor module and produces pixels which are stored in the column buffer. The column buffer is included within the Pixel Buffer module. The incoming voxels have a slant range and depression angle which the Voxel Processor calculated. The depression angle and the voxel's color and opacity are the primary values used when generating pixels.

Voxels entering the Pixel Buffer are accompanied by a color, an opacity, a slant range, a depression angle, and some flags. The Pixel Buffer compares the current depression angle and the previous depression angle to the angular positions of the pixels to determine which whole and partial pixels are covered by this voxel. The whole and partial pixels may be handled two different ways. A bit map may be used to determine the portion of the pixel which has been filled in. This conserves column buffer memory but does not handle translucent pixels well. If a larger column buffer is not a problem, all pixels may be subdivided into subpixels, each of which acts as a single, whole pixel in the column buffer. Within each subpixel, the opaque and translucent sections are accumulated independently with each portion coming in being attenuated by the already accumulated translucent portion. The voxels are accumulated into the pixels in a front to back order. Once an opaque voxel fills in a pixel, the pixel is marked as filled and no further contributions are made to it. As pixels are filled in, the Pixel Buffer maintains the depression angle to the top of the filled-in portion of the column. When that angle reaches the top of the screen the column is finished. Then the pixel column is output to the pixel pipe, with each group of n subpixels being accumulated (averaged) into a single output pixel.

In addition, the Pixel Buffer module performs specularity calculations for the pixels. Since a given voxel, with a specific surface normal, may cover several pixels, the specularity must be computed for each pixel individually. To perform the specularity calculation, the Pixel Buffer uses the depression angle of the pixel and the column angle to compute the vector from the voxel to the eyepoint. Then it uses that vector, along with the vector to the sun, to compute the bisecting vector. Then the cosine of the angle between the bisecting vector and the surface normal is used, along with the specularity, or shininess, of the voxel, to compute the amount of sunlight being reflected toward the eye. The sunlight is then combined with the smooth shaded intensity computed by the pattern processor to give the final color which this voxel imparts to this particular pixel. Note that the opacity of the specular reflection is dependent on the intensity of the specular highlight, not the opacity of the voxel. Thus, transparent glass can have a bright reflective spot which masks anything behind it.

The Pixel Pipe

Once the entire column of pixels is complete, it must pass down the pixel pipe. The first module is the Pixel Mapper which maps each pixel from polar coordinates to flat screen coordinates. The polar coordinates are represented by the ground column angle and the pixel's depression angle. Using the pitch, roll, screen size, and field of view, the pixel is mapped to an (x,y) position on the screen. The following formulas perform the mapping:

$$x' = \frac{(\text{height}/2) * \sin(\text{col\_angle})}{\tan(\text{vfov}/2) * [\cos(\text{pitch}) * \cos(\text{col\_angle}) + \tan(\text{pix\_angle}) * \sin(\text{pitch})]}$$

$$y' = \frac{(\text{height}/2) * [\cos(\text{col\_angle}) * \sin(\text{pitch}) - \cos(\text{pitch}) * \tan(\text{pix\_angle})]}{\tan(\text{vfov}/2) * [\cos(\text{pitch}) * \cos(\text{col\_angle}) + \tan(\text{pix\_angle}) * \sin(\text{pitch})]}$$

Then, the roll angle as applied about the center of rotation as follows:

$$x = x' * \cos(\text{roll}) + y' * \sin(\text{roll}) + \text{x\_center}$$

$$y = -x' * \sin(roll) + y' * \cos(roll) + y\_center$$

For improved performance, the same calculations can be performed with a folded algorithm, as follows:

$$x = \frac{k1 - [k2 * \tan(pix\_angle)]}{k3 + [k4 * \tan(pix\_angle)]} + x\_center$$

$$y = \frac{k6 - [k7 * \tan)pix\_angle)]}{k3 + [k4 * \tan(pix\_angle)]} + y\_center$$

The kn values are subframe and column constants which are computed by the CPG for each column.

The other function of the Pixel Mapper is to implement a haze algorithm. The haze appears as a replacement of some of the pixel's intensity, in IR or RGB, with some haze intensity. The amount of the pixel's intensity remaining is a function of slant range, pitch, and haze density as follows:

$$W_o = e^{Ae^{BZ_{eye}}(1 - e^{R_s B \sin(pitch)})/B\sin(pitch)} \qquad \text{for pitch} \neq 0$$

$$W_o = e^{R_s A e^{BZ_{eye}}} \qquad \text{for pitch} = 0$$

where R_s is slant range from the eye to the pixel, Z_{eye} is the elevation of the eyepoint, pitch is the depression angle of the pixel from the horizon, and A and B are the haze density and vertical distribution parameters, respectively. The haze and original pixel are combined as follows:

$$I_{final} = I_{pixel} * W_o + I_{haze} * (1.0 - W_o)$$

Note that for moving objects it is expected that the haze density will either not vary significantly across them (if they are far away they should be small) or not be very dense (if they are close the haze should be thin) so the pitch to the centroid of the object is used instead of the pixel pitch. This is because the pixel pitch of objects is in terms of the subframe coordinate space, not the terrain coordinate space where the haze exists.

The Frame Buffer module accepts a column of pixels and blasts them into the frame buffer by accumulating the pixel into each of the four corner pixels around the incoming pixel's (x,y) position. The Pixel Mapper module performs the mapping function to a higher degree of accuracy than would be required to simply find the appropriate (x,y) position. The extra resolution is used to determine the mapped pixel's position within the destination pixel. The accumulation into the four neighboring corner pixels is weighted by the inverse of the distance of the mapped pixel position from each corner. This technique is similar to that presented by Westover[12] known as splatting. Each pixel accumulates the sum of the weights of incoming pixels and the sum of the products of weight and intensity. The weighted pixels are then range resolved in a standard Z-buffer implementation.

The frame buffer is the primary place where translucent pixels are combined, although it is also done in the pixel buffer. The algorithm for attenuating colors visible behind a translucent material can be different for different applications. The transport equation defines how background information is attenuated by non-opaque voxels in front. In the visual band (EO), translucent material is treated as an aerosol. Thus, the more opaque the material, the more of it will be seen and the less of the background. If the material is transparent, it provides no color to the pixel at all. In contrast, in the infrared (IR) band, the transmissivity and the emissivity of a material may be relatively independent. Therefore, a material could be totally transparent, and thus pass through all the background contribution, and still emit in the same waveband, thus contributing additional intensity to the pixel. The actual transport equations are:

$$I_p = I_{fg} * O_{fg} + I_{bg} * (1.0 - O_{fg}) \quad \text{for EO}$$

$$I_p = I_{fg} + I_{bg} * (1.0 - O_{fg}) \quad \text{for IR}$$

where I_p is the final pixel intensity, I_{fg} is the foreground material intensity, I_{bg} is the background material intensity, O_{fg} is the opacity of the foreground material, and O_{bg} is the opacity of the background material. Note that the difference in the equations is that the intensity contribution of each material is a function of its own opacity in the EO equation but not in the IR equation.

The Display module normalizes the accumulated intensities in the frame buffer by dividing by the accumulated weights to generate the final image. If the desired image is to be in the visual band, three channels (red, green, and blue) are processed, while a single channel is processed for infrared.

When rendering moving objects, each object is rendered as an individual subframe and the pixel images are range-resolved and merged within the frame buffer. Each object is represented as an individual database and rendered independently. The initialize subframe module transforms the eyepoint in world coordinates and the object position in world coordinates into an eyepoint in object coordinates, relative to the origin and axes of the object database. Multiple objects may be rendered in each frame, and multiple copies of the same object, in different positions, may also be rendered. The objects are range ordered from far to near for rendering so the frame buffer can properly range resolve the pixels. This is because the frame buffer does not store opacities, only range and color. Therefore, if a pixel already has a contribution, it is assumed to be opaque. To ensure that this condition matches the actual pixel ordering, the terrain is rendered first which fills the screen with opaque pixels. Then each object is rendered and range-resolved with the opaque terrain. The objects may have translucent portions which attenuate the background via the transport equation but the attenuated pixel will still be opaque in the frame buffer. Thus, the objects must be rendered from far to near to ensure the proper accumulation of translucent colors. This back to front processing is similar to that presented by Drebin, et al [13].

Anti-aliasing

Aliasing is a significant problem in real-time rendering applications. Problems which are invisible in a single frame become glaring when a sequence of frames are displayed in real time. These temporal aliasing problems include flickering, swimming, strobing, and other anomalies. Of course, spatial aliasing problems, such as jaggies, need to be controlled as well. The Hughes rendering algorithm employs several techniques for controlling both spatial and temporal aliasing. For example, when generating a column of pixels, the pixels are supersampled in the vertical direction and multiple pixels are combined into a single pixel before passing into the pixel pipe. This causes the horizontal edges of objects to be more accurately sampled to prevent popping from one pixel to the next as well as reducing jaggies along the edge.

Another anti-aliasing technique used is called vertical feature anti-aliasing. If a column intersects a vertical feature, such as a wall, at a shallow angle, adjacent voxel columns may intersect the wall several voxels away. Then, as the eyepoint moves, the columns move over to intersect voxels they missed before and the wall appears to shimmy and flicker. This problem is reduced by combining the contributions of all voxels along the wall which the column passes through. The algorithm uses the angle between the column and the surface normal of the wall to determine how many steps it will take for the column to pass from the front of the wall to the back. Then the opacity of the voxel is modified (reduced) proportionately to how many steps remain. The last voxel remains opaque to make sure the wall doesn't appear translucent. This technique is equivalent to supersampling the wall voxels without the cost of extra pixels or columns. Figure 5 illustrates this technique.

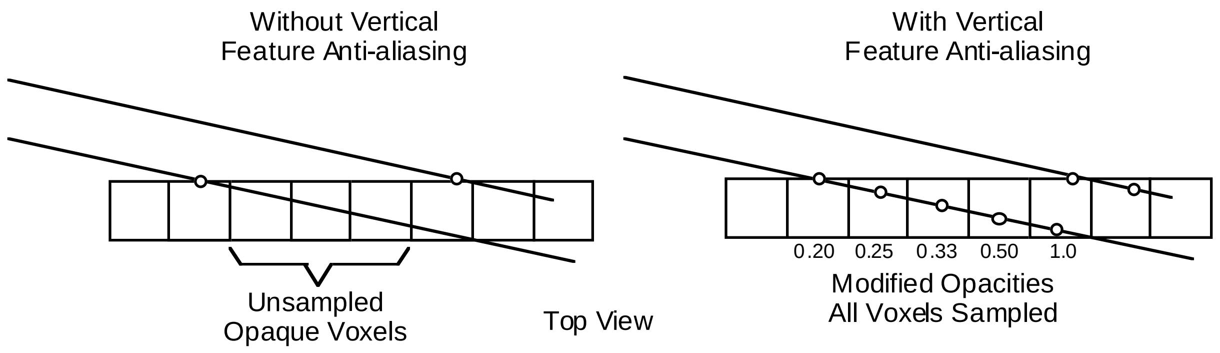

Figure 5 - Vertical Feature Anti-Aliasing

Of course, this technique assumes that the horizontal distance between two columns is less than the width of a voxel. This is handled by the Coordinate Generator by selecting the appropriate resolution voxels based on the slant range from the eye to the voxel. This process also performs an additional anti-aliasing function in that no voxels can be missed by adjacent columns. If voxels could be missed, flickering would occur as voxels were hit and then missed as the eyepoint moved.

Another technique, discussed previously, is blasting, or splatting, in the frame buffer. As a pixel enters the frame buffer, it contributes to four adjacent pixels proportionately to the inverse of its distance from the corners. This prevents edges from popping from one pixel to the next. In addition to blasting, the edges of moving objects, or subframes, are detected and blended with the background pixels for smoother edges with fewer jaggies and less popping.

<u>Results</u>

Here are some examples of images generated using the Hughes rendering algorithm. Figure 6 is a rendered image of Yosemite valley showing Cathedral Rocks. The Yosemite database is courtesy of GeoSpectra Corporation. The terrain resolution is about eight feet per voxel. Figure 7 is a rendered image of Orange County, California, looking east from the ocean. This database is also courtesy of GeoSpectra Corporation. The resolution is approximately 30 meters per voxel with the original image source data being Landsat photos. Figure 8 is a model of the San Onofre nuclear power plant inserted in a terrain database of upstate New York. The power plant model has a resolution of about four feet per voxel while the terrain resolution is about six inches per voxel. The model has been scaled to one-fourth of its normal size, relative to the terrain. Note the smooth shading on the domes. Figure 9 shows a T72 tank model sitting behind a case III tree model. The tank was voxellized at three inch resolution from a polygon model. The tree was grown fractally and then voxellized. Figure 10 shows a view of the Camp Pendleton Marine Base in southern California showing haze. The algorithm has also been used to render smoke cubes, missile plumes, and a rotating helicopter rotor blade.

The Hughes algorithm has been implemented in C, on a Sun SPARCStation 2 with 20 megabytes of memory and running X windows, and in special purpose hardware. The hardware development effort, the REALSCENE™ program, implemented the algorithm using a pipelined architecture virtually identical to figure 4. Each module was implemented as a separate board in a custom backplane chassis. This hardware implementation has been extensively simulated in FORTRAN. The C version is significantly faster than the FORTRAN simulation because it does not try to emulate the low-level hardware details. For a 384 by 384 pixel image, the following rendering times have been recorded:

Sun	272 seconds
Special purpose hardware	0.1 seconds

Note that for this comparison, no case III voxels were present in the database, nor were any moving objects present. For general rendering purposes, the Sun has rendered images in times ranging from less than one minute to up to 12 hours for the same size image. This is dependent on the size of the database and how much page swapping had to be done during rendering. Therefore, direct comparisons of rendering times are not very useful. Suffice it to say that the special purpose hardware is about three orders of magnitude faster than the Sun for this algorithm.

Conclusions

The Hughes rendering algorithm produces high quality images both for terrain and three-dimensional models. It can be implemented in software or special-purpose hardware which can deliver real-time performance. It provides several features considered important for flight simulation applications but is also useful for non-real-time applications, such as mission planning, imagery for hardware in the loop testing, and virtual reality. It can render multiple sensor images of correlated databases with high resolution in the output image. Its pipelineable, modular design suggests that it could perform well on a massively parallel architecture.

Acknowledgements

The authors wish to thank Tom Ellis, Andy Rosman, and many other Hughes people who contributed to the development of this algorithm and the varied implementations of it.

References

[1] Devich, R.N. and Weinhaus, F.M., " Rural Image Perspective Transformations", *Proceedings of SPIE*, vol. 303, 1981, pp. 54-66.

[2] Upson, C. and Keeler, M., " V_BUFFER: Visible Volume Rendering", *Proceedings of SIGGRAPH*, August, 1988, pp. 59-64.

[3] Kaufman, A. and Shimony, E., "3D Scan-Conversion Algorithms for Voxel-Based Graphics", *Proceedings of 1986 ACM Workshop on Interactive 3D Graphics*, October, 1986, pp. 45-76.

[4] Kaufman, A., " Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes", *Proceedings of SIGGRAPH*, vol. 21, no.4, July, 1987, pp. 171-179.

[5] Kaufman, A., "An Algorithm for 3D Scan-Conversion of Polygons", *Proceedings EUROGRAPHICS*, August, 1987, pp. 197-208.

[6] Cohen, D. and Kaufman, A., "3D Scan-Conversion Algorithms for Linear and Quadratic Objects", *Volume Visualization*, IEEE Computer Society Press, 1990, pp. 280-301.

[7] Whiteside, A.E.," Preparing Data Bases For Perspective Scene Generation", *Proceedings of SPIE*, vol. 1075 Digital Image Processing Applications (1989), pp. 230-237

[8] Quarato, J., Brown, S., Chen, C., Nguyen, K., Malpass, J., Thursby, W., and Hester, J.," Voxel Terrain Material Database and Synthetic IR Scene Generation", *Proceedings of Ground Target Modelling and Validation Conference*, April, 1991, pp. 68-88.

[9] Brown, S., Quarato, J., Chang, N., Wright, J., Malpass, J., Thursby, W., and Hester, J.," Voxel Based IR Target Signature and Scene Generation", *Proceedings of Ground Target Modelling and Validation Conference*, April, 1991, pp. 89-103.

[10] Patz, B., Gatt, P., Becker, G., Richie, S., LeBlanc, R., and Coulter, L., " Real Scan Evolution", report prepared for Naval Training Equipment Center, 80-D-0014-2, February, 1982. This report is in the public domain and should be available through the University of Central Florida or the Naval Training Equipment Center.

[11] Foley, J., and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.

[12] Westover, L., " Footprint Evaluation for Volume Rendering", *Proceedings of SIGGRAPH*, August, 1990, pp. 367-376.

[13] Drebin, R., Carpenter, L., and Hanrahan, P., " Volume Rendering", *Proceedings of SIGGRAPH*, August, 1988, pp. 65-74.